RC 20113 (6/26/95) Revised (8/16/95)
Computer Science 10 pages

IBM Research Report

Efficient Data Sharing with Conditional Remote Memory Transfers

Susan Flynn Hummel

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Conditional Remote Memory Transfers

Susan Flynn Hummel [1]

Abstract

Remote memory transfers (RMTs) allow processors to access non-local memories without involving the remote processor. As the only interprocessor synchronization required is setting flags on the local and remote processors after the transfer is complete, RMTs do not entail the overhead due to the handshaking and buffering of message passing nor require the complex hardware of cache-coherent shared memory. The minimal synchronization of RMTs, however, necessiates extra remote communications to enforce common data sharing paradigms, such as producer/consumer. We propose a new hardware mechanism: the conditional RMT (CMRT). CRMTs succeed or fail depending on the state of the remote data. We describe potential CRMT hardware definitions, and illustrate their utility by presenting efficient software protocols for producer/consumer and check-in/check-out data sharing.

1 Introduction

Because it is difficult to maintain the consistency of shared data on large-scale parallel machines, some machines restrict interprocessor communication to message passing, e.g., sends and receives, or, more recently, to remote memory transfers, e.g., puts and gets. Ensuring the consistency of shared data is then the responsibility of the programmer. Remote memory transfers use DMA devices to allow processors to access remote memories without involving the remote processor. In general, accessing non-local data is easier with RMTs than with message passing. Moreover, as the only interprocessor synchronization required is setting flags on the local and remote processors after the transfer is complete, RMTs do not incur as much handshaking and buffering overhead as message passing, nor do they require as complex hardware as cache-coherent shared memory. RMT primitives are supported by the Cray T3D, Fujitsu AP1000 and the Meiko CS-2.

Although the synchronization provided by message passing and cache coherency can be overkill, the minimal synchronization of RMTs necessitates extra remote communications to enforce common data sharing paradigms, such as producer/consumer. We propose a new hardware mechanism: the conditional RMT (CRMT). The basic concept behind CRMTs is that two processors synchronize by having one set a local flag when it is ready for the data, and the other processor check this flag before transferring the data. If the flag is set, then the second processor completes the transfer. If the flag is not set, then the second processor sets a flag on the first processor to let it know that it must complete the transfer. A generalization is to queue processors awaiting transfers.

CRMTs can be implemented in software using active messages [1], which execute a procedure on a remote processor upon arrival rather than waiting for the matching receive to be issued. Active messages, however, are executed by the remote processor. To off-load the execution of active messages to a DMA device would require it to have the full functionality of a general purpose processor. The hardware definitions that we propose for CRMTs are only incrementally more complex than those for RMTs.

Conditional puts and gets can be defined in various ways. We first give a simple definition that uses two boolean flags per processor. We then give a more general definition that

[1] Polytechnic University and IBM T. J. Watson Research Center, email: hummel@mono.poly.edu

uses two integers per processor to implement a queue. We illustrate the utility of the first definition in an implementation of producer/consumer data sharing, and of the second in an implemention of check-in, check-out (CICO) data sharing. In producer/consumer sharing, one processor produces data that are needed by another processor. Check-in and check-out are primitives for changing ownership of shared data; a processor has exclusive access to the data that it (transitorily) owns [2].

2 Data Sharing

Processors executing cooperating subtasks need to share data. In parallel SPMD (single program, multiple data) code, a common data paradigm in *producer/consumer* sharing. In producer/consumer shring, one processor generates a stream of data that are needed by another processor. For example, when High Perfomance Fortran is implemented with an owner computes rule, i.e., updates to an element of a matriix is executed by the processor that owns the element, the follwoing statement in a loop nest can lead to producer/consumer sharing:

```
b(J) = ...
...
a(J) = b(J+1)
```

if the elements of `a` and `b` have been distributed to processor in blocks.
Heterogeneous and dynamic parallelism may require a sharing pattern that is more general than producer/consumer, for example, where data ownership changes over time. Hill *et al.* have proposed a data sharing model called *check-in, check-out* (CICO) that allows processors to transitorally obtain ownership of shared data [2]. At any given time, a single processor (that has successfully) checked out the data in exclusive mode can update the data, or multiple processors (that have successfully) checked out the data in shared mode can read the data.

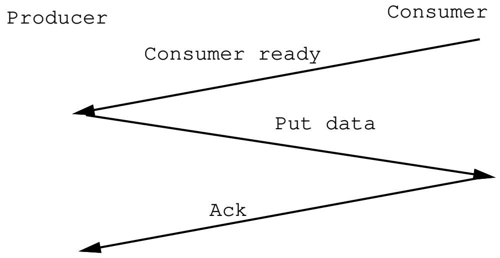

Figure 1: Put-Based Communication for Producer/Consumer Data Sharing

3 Conditional Remote Memory Transfers

Remote memory transfers set flags on the local and remote processors after the transfer is complete. Additional flags are needed to enforce data sharing paradigms, such and producer/consumer [3]. Hardware definitions of puts and gets that use two per-processor flags to enforce producer/consumer sharing have been proposed [3]. The put-based definition is illustrated in Figure 1. Note that three communications take place, one for the actual data transfer and two for handshaking. The acknowledgement is part of the put operation, which sets flags on the local and remote processors when the transfer is complete. As the acknowledgement cannot be piggybacked onto the data transfer, the get-based definition requires an additional communication.

Below we give CRMTs definitions that use 2 flags per-processor. The definitions are very simple, consisting of a test and the setting of the flags.

```
putc(data,f1,f2,result) {
/* The local processor attempts to put data on the remote processor */
  if f1 { /* The remote processor is ready, transfer data */
    f1 = false;
    f2 = false;
    result = true;
    transfer(data);
  } else { /* The remote processor is not ready, signal we were here */
    f2 = true;
    result = false;
  }
}
getc(data,f1,f2,result) {
/* The local processor attempts to get data from the remote processor */
  if f1 { /* The remote processor is ready */
    f1 = false;
    f2 = false;
    result = true;
    transfer(data);
  } else { /* The remote processor is not ready, signal we were here */
    f2 = true;
    result = false;
  }
}
```

In the definitions, `f1` and `f2` are flags on the remote processor, and `result` a flag on the local processor. The remote processor sets `f1` when it is ready, and the local processor sets `f2` to false if it transfers the data. `Result` is used to convey the outcome of the transfer to the local processor.

4 Producer/Consumer

With the hardware definition of CRMTs given in Section 3, software protocols for producer/consumer sharing can be implemented with either 1 or 2 communication operations, depending on whether the producer or consumer is ready first. Protocols can be designed that favor either the producer or the consumer being ready first. In the protocols given below, the producer `p` and consumer `c` each have two flags `pr` (producer ready) and `cr` (consumer ready).

The protocol that favors the producer being ready first is:

```
producer {
cr = false; /* Initial assignments for p */
```

```
pr = false;

  /* If getc happens here it will fail, and set cr to true */
  while (pr) { } /* Wait for previous data to be consumed */
  pr = true;
  /* If getc happens here it will succeed, and set cr and pr to false */
  /* If second getc it will fail and set cr but not pr to true */
  if (cr && pr) {
     = put(c,data,c.pr); /* Sets c.pr to true */
  }
  /* If getc happens here it will succeed, and set cr and pr to false */
  /* If second getc it will fail and set cr to true */
}

consumer {
  pr = false;
  getc(p,data,pr,cr,pr);
  /* If getc fails, pr is set to false, true o.w.  */
  /* If getc fails, then producer will set pr to true */
  while (!pr) { }
}
```

As putc and getc are symmetrical, the protocol that favors the consumer being ready first is similar. The main difference between the protocols is when processors busy-wait.

```
producer {
cr = true; /* Initial assignments for p */

  while (!cr) { } /* cr is set to true by producer or consumer */
  cr = false;
  putc(c,data,cr,pr,cr);
  /* If putc fails, cr is set to false, true o.w.  */
  /* If putc fails, then consumer will set cr to true */
}

consumer {
cr = false; /* Initial assignments for c */
pr = false;

  /* If putc happens here it will fail, and set pr to true */
  cr = true;
  /* If putc happens here it will succeed, and set cr and pr to false */
  /* If second putc, it will fail and set pr but not cr to true */
  if (pr && cr) {
    cr = false;
    get(p,data,cr); /* Sets p.cr to true */
  } /* If putc happens here it will succeed, and set cr and pr to false */
  /* If second putc, it will fail and set pr but not cr to true */
  while (cr) { } /* Wait for data to be produced */
}
```

5 Generalized CRMTs

Data sharing paradigms that are more dynamic than producer/consumer require more complex synchronization. For example, a directory-based protocol has been proposed to coordinate CICO data sharing [2]. If the software adheres to a strict CICO policy, where only data are always checked in before a check out is issued, a check-in takes 3 communications, and a check-out 2 communications. When the CICO policy is validated, a trap is generated and the data are located and forwarded by software.

The definition of CRMTs given in Section 3 use two per-processor flags to coordinate data sharing. CMRTs can be generalized to coordinate CICO data sharing by using two per-processor variables to implement a queue instead of flags: if the data are not ready, then the processor attempting the transfer enqueues its id. When the data become ready, they are forwarded to the processor at the head of the queue. The queue data-structure can be implemented in a variety of ways. Below, we give a linked-list based definition of generalized CRMTs. The number of communications is the same as the directory scheme when a CICO policy is adhered to, and only requires an additional 2 communication when the policy is violated, i.e., when a processor attempts to check out data before it has been checked back in.

```
putc(remote,local,data,f1,f2,result) {
  if f1 != null {
  /* Queue not empty */
    f1 = f1->next;
    if ((f1 == null) or (f1 == remote)) {
    /* Queue now empty or owner is next so return data */
      transfer(data);
      result = local;
    } else result = f1;
  } else {
  /* Queue empty, set f2 as a signal we were here */
    f2 = null;
    result = null;
  }
}

getc(remote,local,data,f1,f2,result) {
  if f1 == null {
  /* Queue empty so get data */
    result = local;
    f1 = local;
    f1->next = null;
    f2 = local;
    transfer(data);
  } else {
    if (f2 == local) result = null;
    else result = f2;
    f2->next = local;
    f2 = local;
```

```
  }
}
```

where f1 and f2 are variables on the remote processor, and result a variable on the local processor.

In the above definitions, f1 and f2 are used as the head and tail pointers of a linked list whose nodes are stored on the remote processor. The queue can be implemented with other data structures, for instance, an array. The definitions for array-based conditional put/get are analogous to the linked-list based ones—f1 and f2 are the first and last indices into an array that are incremented rather than head and tail pointers that follow next pointers. In the next section, we show how to use generalized CRMTs to implement CICO data sharing, and in the appendix, producer/consumer data sharing.

6 Check-in/Check-out

With generalized CRMTs, exclusive check-out can be accomplished with two communication operations, and check-in with two or four communication operations. When checking-out (with a conditional get), if the data are not ready, then the pending owner id is enqueued to let the previous owner know that it must complete the transfer. When checking-in (with a conditional put), the current owner inspects the queue to determine if it must forward the data to a waiting processor. Variations of shared check-outs can be implemented using the proposed CRMTs by having a coordinating processor check-out the data exclusively and forwarding copies to the processors waiting on its queue.

In the code given below, the permanent owner o of the data has two variables head and tail, and each transient owner id has a local variable result.

```
o->head = null; /* Initial assignments */
o->tail = null;

check_out {
  result = null;
  getc(o,id,data,head,tail,result);
  while (result != id) { } /* Set by getc or subsequent put */
}

check_in {
  result = null;
  putc(o,id,data,head,tail,result);
  while(result == null) {}
  if (result != id) put(result,id,data,result);
  /* put sets result->result to result */
}
```

7 Conclusions

We have described a minimal modification to put/get hardware, the conditional put/get, that allows for efficient producer/consumer data sharing. A more significant to the DMA hardware modification allows for efficient check-in, check-out data sharing. The synchronization provided by conditional remote memory transfers also simplifies dynamic load balancing schemes where work is pushed to lightly loaded processors.

Generalized Producer/Consumer

Producer/consumer sharing can be implemented using the generalized CRMTs defined in Section 5. We first give a protocol that favors the producer being ready first. In the protocol, the "queue" is used to coordinate data consumption. As it is local to the producer, while the producer is not ready, the queue is not empty, i.e., the data is checked-out. If the consumer attempts to get the data before the producer sets the queue to empty, i.e., the producer sets head to null, then the consumer fails and enqueues itself. The producer can check for this later condition.

```
producer {
head = c; /* Initial assignments for p */
tail = c;
c->next = null;

  /* If getc happens here it will fail, and set c->next to c */
  while (head == null) { } /* Wait for previous data to be consumed */
  head = null;
  /* If getc happens here it will succeed, and set head to c */
  /* If second getc happens it will fail, and set c->next to c */
  if ((c->next != head) && head == null)) {
  /* First getc failed, so complete transfer */
  /* Consumer is busy waiting on result so safe to update */
    c->next = null; /* Re-initialize variables */
    tail = c; /* Not strictly necessary as we never inspect */
    head = c;
    put(c,p,data,result); /* Sets result to c */
  }
  /* when getc happens it will succeed, and set head to c */
  /* If second getc happens it will fail, and set c->next to c */
}

consumer {
  result = null;
  getc(p,c,data,head,tail,result);
  /* If getc fails, result is set to null, c o.w.  */
  /* If getc fails, then producer will set result to c */
  while (result != c) { }
}
```

The structure of the protocol that favors the consumer being ready first is similar to the protocol that favors the producer. The differences in the protocols are due to the different locations of queue (producer or consumer) and the different times that the consumer/producer must busy wait. In the putc protocol, the queue is used to coordinate data production. As it is local to the consumer, while the consumer is not ready, the queue is empty—there are no pending resuests. If the producer attempts to put the data before the consumer has enqueued itself, i.e., the consumer sets `head` to `c`, then the producer fails and sets `tail` to `null`. The consumer can check for this later condition.

```
producer {
result = p; /* Initial assignment for p */
  while (result != p) { } /* result is set to p by producer or consumer */
  result = null;
  putc(p,c,data,head,tail,result);
  /* If putc fails, result is set to null, p o.w.  */
  /* If putc fails, then consumer will set result to p */
}
consumer {
head = null; /* Initial assignments for c */
tail = c;
c->next = null;
  /* If putc happens here it will fail, and set tail to null */
  head = c;
  /* If putc happens here it will succeed, and set head to null */
  /* If second putc happens it will fail, and set tail to null */
  if ((tail != head) && (head == c)) {
  /* First putc failed, so complete transfer */
  /* Producer is busy waiting on result so safe to update */
    head = null; /* Re-initialize variables */
    tail = c;
    c->next = null;
    get(p,c,data,result); /* Sets result to p */
  }
  /* When putc happens it will succeed, and set head to null */
  /* If second putc happens it will fail, and set tail to null */
  while (head != null) { } /* Wait for data to be produced */
}
```

References

[1] T. von Eickein, D. Culler, S. Goldstein and K. Shauser, Active Messages: a Mechanism for Integrated Communication and Computation, *Proc. 19th Int. Sym. on Computer Architecture* pp. 256-266, May 1992.

[2] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors, *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems* pp. 262-273, Oct. 1992.

[3] M. Gupta and E. Schonberg, Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs, IBM RC, 1995.